Multimorphic Testing

Temple, Paul; Acher, Mathieu; Jézéquel, Jean-Marc

Publication date:
2018

Document Version
Publisher's PDF, also known as Version of record

Multimorphic Testing

Paul Temple, Mathieu Acher, Jean-Marc Jézéquel

Multimorphic Testing

Paul Temple
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
paul.temple@irisa.fr

Mathieu Acher
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
mathieu.acher@irisa.fr

Jean-Marc Jézéquel
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
jean-marc.jezequel@irisa.fr

ABSTRACT

The functional correctness of a software application is, of course, a prime concern, but other issues such as its execution time, precision, or energy consumption might also be important in some contexts. Systematically testing these quantitative properties is still extremely difficult, in particular, because there exists no method to tell the developer whether such a test set is "good enough" or even whether a test set is better than another one. This paper proposes a new method, called *Multimorphic testing*, to assess the relative effectiveness of a test suite for revealing performance variations of a software system. By analogy with mutation testing, our core idea is to vary software parameters, and to check whether it makes any difference on the outcome of the tests: i.e. are some tests able to "kill" bad morphs (configurations)? Our method can be used to evaluate the quality of a test suite with respect to a quantitative property of interest, such as execution time or computation accuracy.

ACM Reference Format:
Paul Temple, Mathieu Acher, and Jean-Marc Jézéquel. 2018. Multimorphic Testing. In *ICSE '18 Companion: 40th International Conference on Software Engineering Companion, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

On May 7, 2016, a 2015 Tesla Model S collided with a tractor trailer crossing an uncontrolled intersection on a highway west of Williston, Florida, USA, resulting in fatal injuries to the Tesla driver. On January 19, 2017, the NHTSA (National Highway Traffic Safety Administration) released a report on the investigation of the safety of the Tesla autonomous vehicle control system. Data obtained from the Model S indicated that: 1) the Tesla was being operated in Autopilot mode at the time of the collision; 2) no actions or warning signals were initiated neither from the driver nor the Automatic Emergency Braking system. The conclusion was the investigation did not reveal any safety-related defect with respect to predefined requirements from the system.

However, the crash did actually occur. Without questioning the legal aspects that are definitively covered in the NHTSA report, one might wonder why the computer vision program did not "see" this huge trailer in the middle of the road. Of course, a posteriori, it is easy to understand that the Tesla crash videos recorded by Autopilot were not under ideal lighting conditions. Background objects blended into vehicles that needed to be recognized, making it difficult for any computer to process the video stream correctly. On top of that, no wheels were visible under the trailer, which complicated its identification as a vehicle in the middle of the road.

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Now taking a software engineering perspective, how come that this situation has not been tested before the software was deployed? Put this way, this is, of course, a familiar question to any tester, with the usual (difficult) answer involving a huge input data space. Since the input here is video, it is even orders of magnitude larger than typical data. Beyond the Tesla case, there are several software applications for which (1) quantitative properties (execution time, energy consumption, *etc.*) are crucial; (2) the acceptable behavior of a program is hard to characterize; (3) the size of the input space (test case) is enormous. This leads to a set of questions: How can we build a "good" test suite? Depending on the software application's goals (*e.g.,* maximizing speed, computations' accuracy or even a tradeoff among several such quantitative properties), do we end up with the same "good" test set? How do we even know that a given test suite is "better" than another one? Structural code coverage metrics for test suites seem indeed a bit shaky, especially to handle performance aspects and quantitative properties. To our knowledge, no method exists to assess the "coverage" of test suites with respect to their ability to reveal performance weaknesses in the software.

In this paper, we propose a method to assess the relative value of test suites. By analogy with mutation testing [2, 4, 5], the core idea of multimorphic testing is to synthesize morphs (*e.g.,* through the variations of parameters' values), execute test cases over morphs and finally check differences on the outcome of tests: *i.e.,* are some test cases able to "kill" program's configurations? "Killing" intuitively means exhibit high variations w.r.t. quantitative properties of the morphs (*e.g.,* some morphs are too slow).

2 MULTIMORPHIC TESTING

Principle. Multimorphic testing proactively produces system variants that act as competitors to an original system. Leveraging Software Product Line automatic derivation techniques [1, 3], we create multiple *morphs* (see variants $S_2, S_3, \ldots, S_n$ in Figure 1) for the purpose of comparatively confronting and assessing their performances w.r.t. a given test set (see performance matrix). In the end, we consider a test is "good" when it is able to reveal significant quantitative differences in the behavior of variants of a system. Pursuing the mutation testing analogy: we derive and exploit variants to assess that tests are able to "kill" them by exhibiting significant quantitative differences (instead of pass/fail verdicts) and eventually assess the quality of a test suite.

Figure 1 gives an overview of the different steps. Our method assumes that test cases $T_1, T_2, ..., T_m$ exist. The output of multimorphic testing is a measurement of the quality of a test suite. The first step of our process, called *multimorphing*, consists in producing variants of a system (see the top of Figure 1). Numerous mechanisms can be employed for varying a program (mutation operators, approximate computation or variability techniques, *etc.*) which make our framework general. We can typically exploit the

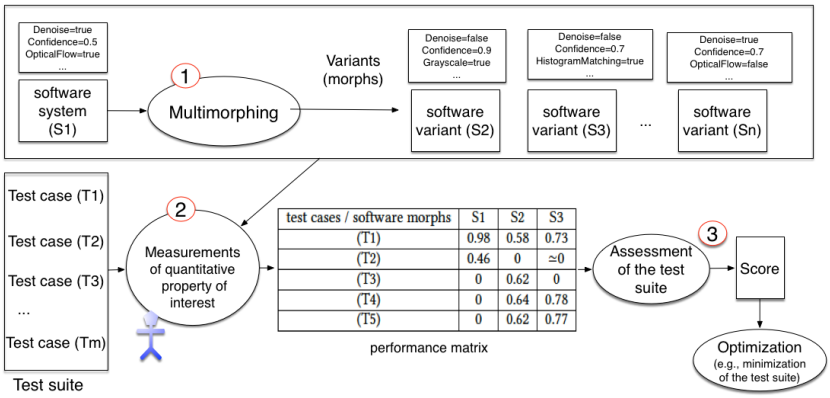

Figure 1: Multimorphic testing: the goal is to assess the ability of a test suite in revealing performance variations

various program parameters to model its variability and automatically derive different systems $S_2, S_3, ..., S_n$ out of configurations (see right-hand side of Figure 1). By doing so, we guarantee to preserve the coarse-grained functionality of a program since parameters only have an effect on quantitative properties (*e.g.,* execution time, etc.). For instance, Denoise is set to false for the variants S_2 and S_3. On one hand, it can save time since no computations are needed to remove noise. On the other hand, this parameter can have negative effects in terms of computations' accuracy. In any case, all variants $S_2, S_3, ..., S_n$ realize the functionality; we only expect differences in terms of quantitative properties such as execution time due to different parameterization. The second step of the process (see the bottom of Figure 1) is to execute all systems $S_1, S_2, S_3, ..., S_n$ over test cases $T_1, T_2, ..., T_m$. We measure quantitative properties of interest for each pair morph/test case *e.g.,* How accurate are computations? How much times does it take? Numerous properties can be individually considered and combined. It is up to users of our method to define their property of interest. Based on executions, we obtain a *performance matrix* of variants over test cases.

Let us take an example and consider the matrix of Figure 1 (at the bottom center). On columns, different program variants, obtained through the settings of parameters' values, are presented. Rows represent test cases that have been gathered. For each pair program/test case, performance measures are reported. We can notice two important phenomena in this matrix. First, each system exhibits important performance variations due to differences in provided input test cases. For instance, considering S_1, its performances for the first test case (0.98) are significantly different from performances for the fourth test case (0). Similar observations considering other morphs can be made over the rest of the matrix. This example first shows that test cases can severely impact the precision measures of individual programs. The selection of test cases is therefore crucial since their inclusion in the test suite can have a dramatic impact over observations of variants' behavior. Second, considering a given test case, program variants exhibit performance variations. For test cases $T1$ and $T4$, precision varies for the three programs. For $T2$ and $T3$, the situation is a bit different since two measurements are the same (*i.e.,* 0). The example shows that the synthesis of program variants reveals new insights on the quality of a test suite. That is: some test cases exhibit quantitative differences within morphs.

Thanks to the synthesis of multiple variants, we can determine the impact of test cases regarding the relative performances of programs (third step). We propose to use the notion of *dispersion score* to assess whether observations spread over a range and somehow "cover" this range. Several methods can be considered here and are out of the scope of this poster. We can then envision to build an optimal test suite (*e.g.,* that would remove weak test cases).

3 CONCLUSION AND FUTURE WORK

We proposed a new method, called *Multimorphic testing*, to assess the effectiveness of a test suite in revealing performance weaknesses of a program. Our method can be applied for various quantitative properties of programs such as computation' accuracy, execution time, or their compositions. The core idea of multimorphic testing is to vary the program parameters and to check whether it makes any difference on the outcome of the tests in terms of the quantitative property of interest. Intuitively a "good" test has a good discriminating power over the set of program variants. Thanks to our method, we can envision to remove unnecessary, redundant test cases or improve existing test data sets.

Future work includes investigating our method in different application domains (such as computer vision, compilers and generators) where quantitative properties are of prior importance.

Acknowledgments. This work benefited from the support of the project ANR-17-CE25-0010-01 VaryVary.

REFERENCES

[1] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. 2013. FAMILIAR: A domain-specific language for large scale management of feature models. *Science of Computer Programming (SCP)* 78, 6 (2013), 657–681.
[2] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. 2006. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *IEEE Transactions on Software Engineering* 32, 8 (Aug 2006), 608–624.
[3] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer-Verlag.
[4] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. 2013. Comparing Non-adequate Test Suites Using Coverage Criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013).* ACM, 302–313.
[5] José Miguel Rojas, Mattia Vivanti, Andrea Arcuri, and Gordon Fraser. 2017. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering* 22, 2 (2017), 852–893.